Assignment 2

Derek Paulsen

1 Problem 1

1.1 Part A

We prove by induction on the depth of the tree. We show that we can force the algorithm to evaluate the tree with a min or max node at the root, regardless of the final value of the tree.

Base Case

Suppose we have a tree of height 1 with a max node at the root, with two children, c_1, c_2, which are leaf nodes. Because the algorithm is deterministic we know which child we will be searched first, WLOG we assume that c_1 is searched first. We then set the value of c_1 to be 0, which then forces the algorithm to search c_2. We then set c_2 to be the value of the tree, i.e. 0 if the tree's value is 0 otherwise 1. Note that we can force the algorithm to search both children regardless of the actual value of the subtree.
Suppose we have a tree of height 1 with a min node at the root, with two children, c_1, c_2, which are leaf nodes. Because the algorithm is deterministic we know which child we will be searched first, WLOG we assume that c_1 is searched first. We then set the value of c_1 to be 1, which then forces the algorithm to search c_2. We then set c_2 to be the value of the tree, i.e. 0 if the tree's value is 0 otherwise 1. Note that we can force the algorithm to search both children regardless of the actual value of the subtree.
Thus we can force the algorithm to search both nodes, regardless of the value of the tree, for both min and max nodes. This completes the base case.

Inductive Step

Now suppose that we have a max node at the root of the tree of height h with two children, c_1, c_2. Because the algorithm is deterministic we know which child we will be searched first, WLOG we assume that c_1 is searched first. We then set the value of c_1 to be 0. Because c_1 is a tree of height $h-1$, by the inductive hypothesis we can force the algorithm to evaluate the entire subtree regardless of the value of the subtree. We then set c_2 to be the value of the tree, i.e. 0 if the tree's value is 0 otherwise 1. Again, Because c_2 is a tree of height $h-1$, by the inductive hypothesis we can force the algorithm to evaluate the entire subtree regardless of the value of the subtree. Therefore we can force the algorithm to evaluate every node in the tree of height h regardless of the value of the tree.
Now suppose that we have a min node at the root of the tree of height h with two children, c_1, c_2. Because the algorithm is deterministic we know which child we will be searched first, WLOG we assume that c_1 is searched first. We then set the value of c_1 to be 1. Because c_1 is a tree of height $h-1$, by the inductive hypothesis we can force the algorithm to evaluate the entire subtree regardless of the value of the subtree. We then set c_2 to be the value of the tree, i.e. 0 if the tree's value is 0 otherwise 1. Again, Because c_2 is a tree of height $h-1$, by the inductive hypothesis we can force the algorithm to evaluate the entire subtree regardless of the value of the subtree. Therefore we can force the algorithm to evaluate every node in the tree of height h regardless of the value of the tree.
This we have forced the algorithm to evaluate the entire tree if there is a min, or max node at the root, regardless of the final value of the tree. This completes the inductive step.

1.2 Part B

We outline or search procedure below. At each step we maintain the worst case for the min and max player, u and l, respectively. We then use these bounds to skip evaluating subtrees when we find that $l \geq u$ meaning that neither player will choose this subtree.

Algorithm

```
def search(node, l, u):
    if node.is_leaf():
        return node.score

    # randomly choose which child to search first and second
    first_child, second_child = shuffle(node.children)

    if node.is_max():
        score = search(first_child, l, u)
        l = max(l, score)
        # this subtree is not competitive, skip evaluating the rest
        if l >= u:
            return score
        else:
            return max(score, f(second_child, l, u))

    if node.is_min():
        score = search(first_child, l, u)
        u = min(u, score)
        # this subtree is not competitive, skip evaluating the rest
        if l >= u:
            return score
        else:
            return min(score, f(second_child, l, u))
```

We then call this search function with the following

```
# start at the root, worst case for max player is -infinity and
# worst case for min player is +infinity
game_value = search(gametree.root, -inf, +inf)
```

Analysis

We begin by noting that there are two cases at each step, either we search the sub-tree that has the true value of the node first or second. In the former case, we have a good bound, meaning that the true value of the second subtree is worse that the first. In this case there are two options. Either on subtree has the true (worse) value or both subtrees have worse values. This means that we have a 75% chance of skipping evaluating the second subtree. In the latter case, our bound is not good, hence we need to evaluate both subtrees. This gives us the following recurrence.

$$\begin{aligned} b_i &= \frac{3}{2}b_{i-1} + \frac{1}{2}g_{i-1} \\ g_i &= \frac{1}{4}b_{i-1} + g_{i-1} \end{aligned}$$

We can then get g_i in terms of b_i,

$$\begin{aligned} g_i &= \frac{1}{4}b_{i-1} + g_{i-1} \\ g_i &= \frac{1}{4}(b_{i-1} + b_{i-2}) + g_{i-2} \\ g_i &= \frac{1}{4}(b_{i-1} + b_{i-2}, ...) + 1.25 \\ g_i &= \frac{1}{4}\sum_{j=1}^{i-1} b_j + 1.25 \end{aligned}$$

We can then subsitute this, into the other recurrence,

$$
\begin{aligned}
b_i &= \frac{3}{2}b_{i-1} + \frac{1}{2}g_{i-1} \\
b_i &= \frac{3}{2}b_{i-1} + \frac{1}{8}(\sum_{j=1}^{i-2} b_j + 1.25) \\
b_i &= \frac{11}{8}b_{i-1} + \frac{1}{8}(\sum_{j=1}^{i-1} b_j + 1.25) \\
b_i - b_{i-1} &= \frac{11}{8}(b_{i-1} - b_{i-2}) + \frac{1}{8}(\sum_{j=1}^{i-1} b_j + 1.25) - \frac{1}{8}(\sum_{j=1}^{i-2} b_j + 1.25) \\
b_i - b_{i-1} &= \frac{11}{8}(b_{i-1} - b_{i-2}) + \frac{1}{8}b_{i-1} \\
b_i &= \frac{5}{2}b_{i-1} - \frac{11}{8}b_{i-2}
\end{aligned}
$$

We then get the characteristic equation for this recurrence, which we then solve for the roots,

$$
\begin{aligned}
0 &= x^2 - \frac{5}{2}x + \frac{11}{8} \\
x &= 1.683 \\
x &= 0.817
\end{aligned}
$$

This then gives us the formula for b_i,

$$b_i = x1.683^i + y0.817^i$$

Where x and y are some constants. Since $i = \log_2(n)$ (the number of leaves we then get that the algorithm for a tree with n leaves requires,

$$
\begin{aligned}
\text{leaves evaluated} &= x1.683^{\log_2(n)} + y0.817^{\log_2(n)} \\
\text{leaves evaluated} &= xn^{\log_2(1.683)} + yn^{\log_2(0.817)} \\
\text{leaves evaluated} &= xn^{\log_2(1.683)} + yn^{\log_2(0.817)} \\
\text{leaves evaluated} &= O(n^{\log_2(.753)})
\end{aligned}
$$

therefore the algorithm runs in $O(n^{\log(.753)})$ time.

2 Problem 2

2.1 Part A

In order for the game tree evaluation to fail we must have at least one node evaluate to the wrong value (i.e. return 0 when the value is actually 1). If we query each leaf $2\log n$ times (resulting in a total of $2n\log n$ leaf evaluations) we get,

$$
\begin{aligned}
Pr[\text{the leaf fails}] &\leq 1/3^{2\log n} \\
Pr[\text{the leaf fails}] &\leq n^{2\log(1/3)} \\
Pr[\text{the leaf fails}] &\leq n^{\log(1/9)}
\end{aligned}
$$

We can then union bound the probability of at least one leaf fails

$$\begin{aligned}
&Pr[\text{at least one leaf fails}] \leq \sum_{i=1}^{n} n^{\log(1/9)} \\
&Pr[\text{at least one leaf fails}] \leq nn^{\log(1/9)} \\
&Pr[\text{at least one leaf fails}] \leq n^{1+\log(1/9)} \quad (1+\log(1/9) = -1.197) \\
&Pr[\text{at least one leaf fails}] \leq 1/n
\end{aligned}$$

Since the probability of a single leaf failing is greater than or equal to the game tree evaluating to the incorrect value we have that the probability that the game tree evaluates to the correct value is less than or equal to $1 - 1/n$.

2.2 Part B

Our algorithm is as follows,

```
EvaluateTree(node):
        if node.is_max():
                for c in children:
                        c.value = EvaluateTree(c)

                        if c.value == 1:
                                return 1

        if node.is_min():
                for c in children:
                        if c.value != 1:
                                c.value = EvaluateTree(c)

                return min(children.values)
```

Essentially, all we do is query each subtree in a round robin fasion until the value of the current node is 1, skipping any children that have already been evaluated.
We prove this statement inductively,

2.2.1 Base Case

Suppose we have a max node with k leaf nodes for children. Additionally, suppose that the true value of the node is 1 and only a single child node's true value is 1. If we query every child once, repeating until one child returns 1, the expected number of queries is $1.5k$ since there is one child which has true value one and the expected number of evaluations for that child to return 1 is 1.5. Note that this is a worst case, if more children have a value of 1, the expected number of queries is less.
Suppose we have a min node with k leaf nodes for children. Additionally, suppose that the true value of the node is 1, If we query every child once, repeating until every child returns 1 (skipping leaves that have already returned 1), since every child's true value is 1 we expected each child to be queried 1.5 times, by linearity of expectation, the expected number of queries for the min node to evaluate to 1 is $1.5k$.
Hence we have shown that that for any tree of height 1 with k leaves we only require $1.5k$ leaf evaluations in order to discover that the value of the node is in fact 1, however if the node's value is actually 0, the first round of queries will evaluate to the true value of the tree.

2.2.2 Inductive Case

Suppose we have a max node which has a true value of 1. Again we assume that only one child has a true value of 1. If we apply the same iterative querying procedure as above (where we query each leaf exactly once per round), by induction the expected number of rounds until a child evaluates to 1 is 1.5 rounds, if the total number of leaves in the entire subtree is k then the total number of leaf evaluations is $1.5k$ expectation.
Suppose we have a min node which has a true value of 1. . If we apply the same iterative querying procedure as above (where we query each subtree exactly once per round if it hasn't already evaluated to 1), by induction

the expected number of rounds until all child evaluate to 1 is 1.5 rounds, resulting in a $1.5k$ leaf evaluations in expectation.
By induction we then have that with the iterative querying procedure above we require $1.5n$ leaf evaluations in expectation to learn the true value of the game tree given that the value is 1. Note that if the true value of the game tree is 0, the number of expected number of evaluations required to learn the true value is n since having some leaves return 0 erroneously doesn't change the final value to the tree in this situation. Let X be the number of evaluations required to learn the true value of the game tree, by Markov's inequality to get,

$$Pr[X \geq tn] \leq 1.5n/tn$$

We can then set $t = 150$ to get

$$\begin{aligned} Pr[X \geq 150n] &\leq 1.5n/150n \\ Pr[X \geq 150n] &\leq 1/100 \end{aligned}$$

Therefore with $150n = O(n)$ leaf evaluations we can learn the true value of the game tree with at least 99% probability.

3 Problem 3

3.1 Part A

3.2 Part B

Our algorithm is as follows, let $S = \{1, ..., n^5\}$. For each edge (u, v) assign randomly assign a value to $x_{u,v}$ from S. Next, take the determinant of the resulting matrix A, if $det(A) \neq 0$ there exists a perfect matching, otherwise there is no perfect matching.

3.2.1 Runtime

Constructing the matrix A takes n^2 operations and computing $det(A)$ takes n^3 operations, therefore the total runtime is $O(n^3)$.

3.2.2 Probability

We note that $det(A)$ is a polynomial of degree at most n. From the Schwartz-Zippel Theorem, we have that $Pr[det(A) = 0] \leq n/k$. Here $k = |S| = n^5$, therefore we have $n/k = 1/n^4$. Therefore our probability of success is at least $1 - 1/n^4$.

3.3 Part C

We can use the above algorithm to compute a perfect matching as follows.
Let $f(V, E) \rightarrow \{true, false\}$ be the procedure above, returing $true$ if there exists a perfect matching.

```
• FindPerfectMatching($V, E$)
• If $f(V, E) = false$ return $\emptyset$
• If $|V| = 0$ return $\emptyset$ // all vertexes paired
• For $(u, v) \in E$
    – $V' \leftarrow V \setminus \{u, v\}$ // remove the paired vertexes
    – $E' \leftarrow \{(x, y) | (x, y) \in E \wedge u, v \notin (x, y)\}$ // remove the edges for the matched vertexes
    – If $f(V', E') = true$ return $\{(u, v)\} \cup$FindPerfectMatching($V', E'$)
```

3.4 Part D

Using our randomized algorithm we reduce the problem of finding a maximum matching to finding the largest sub matrix with a non-zero determinant. We then note that the largest sub matrix with non-zero determinant is the $rank(A)$. We can then use Gaussian elemination to get the matrix into reduced row echelon form. We then know the $rank(A) = k$. Let I indexes of the columns of A in reduced row echelon form which are non-zero. Since I is the set of vertexes that participate in a matching of size k. Hence we get the following algorithm,

- FindMaximumMatching(V, E)
- $A \leftarrow$ the randomized tutte matrix from above
- $A' \leftarrow A$ in reduced row echelon form
- $V' \leftarrow$ the indexes of the non-zero columns of A'
- $E' \leftarrow \{(u, v) | (u, v) \in E \wedge u, v \in V'\}$
- return FindPerfectMatching(V', E')

4 Problem 4

4.1 Part A

All of the nodes that can reach v can now reach u and vice versa. Mathmatically, we update rows a_u and a_v as follows,

$$i = 1, ..., n \quad A_{v,i} = max(A_{u,i}, A_{v,i}) \quad A_{u,i} = max(A_{u,i}, A_{v,i})$$

In other words the rows are updated to be the bitwise or of the two rows.

4.2 Part B

Let X be the vector $(x^1, ...x^n)$. For each row we construct a Fenwick Tree. Which keeps track of the prefix sums for $a_u \times X$ (the pairwise product of the row with the polynomial).

Fenwick Trees can compute the sum of the first k elements in $O(\log n)$ time, hence we compute $p(x; a, b) = p(x; 1, b) - p(x; 1, a)$, which takes $O(2 \log n) = O(\log n)$ time

Fenwick Trees can be updated in $O(\log n)$ if a single element changes, hence if we have k elements change we can update the Fenwick Trees in $O(k \log n)$ time.

4.3 Part C

We note that to find the elements that need to be changed we want to compute the symmetric difference of rows a_u and a_v. We do this as follows,

```
Update(A, (u, v))
  1. While $sum(a_u) \neq sum(a_v)$
     (a) $i \leftarrow$ FindDiff($a_u, a_v, 1, n$)
     (b) If $A_{u,i} \neq 1$ update row $u$ and the Fenwick Tree for row $u$
     (c) Else update row $v$ and the Fenwick Tree for row $v$
```

Our binary search procedure is then,

FindDiff(a_u, a_v, l, u)

```
1. FindDiff($a_u, a_v, l, u$)
2. if $l = u$ return $l$
3. $m \leftarrow (l + u)/2$
4. If $sum(a_u[l..m]) \neq sum(a_v[l..m])$ return FindDiff($a_u, a_v, l, m$)
5. Else return FindDiff($a_u, a_v, m, u$))
```

We note that FindDiff succeeds with high probability if we choose x from a sufficiently large set S.

4.3.1 Runtime

Our while loop in update procedure runs k times since it finds a single element each loop. It takes $O(\log n)$ calls to FindDiff, each of which requires $O(2 \log n)$ time to compute the two sums of the sub arrays therefore it takes $O(\log^2 n)$ time to find a single element to update. finally, it then does a single update which is $O(\log n)$ time. Therefore we have the total runtime to be $O(k(\log^2 n + \log n)) = O(k \log^2 n)$

4.4 Part D

Finally, we simply update matrix A iteratively using the procedure above. Each iteration requires one check of the sums which requires $O(\log n)$ time to compute. There are T iterations hence the initial check requires, $O(T \log n)$ time. If the check fails (there needs to be an update) then we run the update procedure k times. For any sequence of edges we update at most $n^2 - n$ entries, therefore the updating requires $O(n^2 \log^2 n)$ time. Summing these together we get that the runtime is $O(T \log n + n^2 \log^2 n)$.